Formal calculation unifying engineering theories beyond software

Raymond Boute, INTEC, Universiteit Gent, B-9000 Gent (Belgium)

Marktoberdorf, 2004/08/13

Abstract

There are rifts at various levels between classical engineering (electrical, mechanical, civil) and software engineering, caused by methodological differences between the underlying mathematics, in particular calculus and logic. Formal calculation, in the sense of expression manipulation on the basis of syntax ("letting the symbols do the work") provides a very effective means of unification.

In this lecture, a formalism is presented that implements this principle. It consists of a very simple language that is free of all defects present in common mathematical conventions, together with a collection of rules that makes formal calculation suitable for everyday practice throughout engineering mathematics.

The main rules are those derived for so-called *generic functionals* and for a *functional predicate calculus*. The resulting unification is illustrated by a series of examples ranging from classical engineering (related to analysis and systems theory) to computing science (data types, relational data bases, various theories of programming) and topics comon to both (automata).

Keywords Computing Science, Concrete Generic Functionals, Data Bases, Formal Calculation, Functional Predicate Calculus, Programming Theories, Quantifiers, Software Engineering, Systems Theory, Unification

1 Introduction

1.1 Motivation: rifts between engineering theories

As Parnas [30] observes, "professional engineers can often be distinguished from other designers by the engineers' ability to use mathematical models to describe and analyze their products". This situation exists *de facto* in classical engineering areas (electrical, mechanical, civil etc.), where the use of mathematics (algebra, calculus etc.) is routine.

By contrast, current software design practice hardly uses mathematical models, save perhaps for some critical applications. The special name "formal methods" for designating mathematical techniques in software confirms their exceptional status [16]. Software is the area of engineering that least exploits available scientific methods.

The usual arguments that the mathematics needed is too hard for "the average software designer" or not suitable in practice do not provide a satisfactory explanation. Indeed, classical engineering mathematics is no less demanding, and application to practical problems requires at least an equal degree of preparation and insight.

An important cause of the disparity is that, on one hand, classical engineers are generally better prepared, yet, on the other hand, software engineering requires mathematics closer to formal logic, which is not part of classical engineering mathematics.

The crucial difference is not in content but in style and method: traditional formal logic has neither the practical usefulness by the standards of Dijkstra [15] or Gries [18] nor the elegance engineers appreciate in applied mathematics (see e.g. Parnas [29]).

Formula manipulation in classical engineering mathematics, such as for derivatives and integrals in calculus, is essentially *formal*, i.e., guided by the shape of the expressions. Here are examples taken from two typical engineering textbooks, one on Fourier transforms by Bracewell [11], another on coding by Blahut [2].

$$
\begin{aligned}
F(s) &= \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi xs} dx \\
&= 2\int_{0}^{+\infty} e^{-x} \cos 2\pi xs \; dx \\
&= 2 \text{ Re} \int_{0}^{+\infty} e^{-x} e^{i2\pi xs} dx \\
&= 2 \text{ Re } \frac{-1}{i2\pi s - 1} \\
&= \frac{2}{4\pi^2 s^2 + 1} \qquad (1)
\end{aligned}
$$

$$
\begin{aligned}
\frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) l_n(\mathbf{x}) & \\
&\leq \frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta)[1 - \log q^n(\mathbf{x})] \\
&= \frac{1}{n} + \frac{1}{n} L(\mathbf{p}^n; \mathbf{q}^n) + H_n(\theta) \\
&= \frac{1}{n} + \frac{1}{n} d(\mathbf{p}^n, \mathcal{G}) + H_n(\theta) \\
&\leq \frac{2}{n} + H_n(\theta) \qquad (2)
\end{aligned}
$$

The typical formal rules used are the rules for arithmetic (associativity, distributivity etc.) plus those from calculus and those for the functions in the area of discourse.

By contrast, logical arguments supporting such rules are nearly always presented informally. In a comment about the first chapter of his book [34], Taylor observes

> The notation of elementary school arithmetic, which nowadays everyone takes for granted, took centuries to develop. There was an intermediate stage called *syncopation*, using abbreviations for the words for addition, square, root, *etc.* For example Rafael Bombelli (*c.* 1560) would write
>
> R. c. L. 2 p. di m. 11 L for our $\sqrt[3]{2 + 11i}$.
>
> Many professional mathematicians to this day use the quantifiers ($\forall$, $\exists$) in a similar fashion,
>
> $\exists \delta > 0$ s.t. $|f(x) - f(x_0)| < \epsilon$ if $|x - x_0| < \delta$, for all $\epsilon > 0$,
>
> in spite of [...] Even now, mathematics students are expected to learn complicated (ϵ-δ)-proofs in analysis with no help in understanding the logical structure of the arguments. Examiners fully deserve the garbage that they get in return.

Clearly the rift between classical and software engineering mirrors the style breach between formula manipulation and logical arguments present in most areas of mathematics. This style breach also exists *within* theories for software: even when the subject matter is essentially formal, say, formal program semantics [26, 27, 32, 33, 35, 37], the logical developments of the underlying theories are (with rare exceptions [14, 17]) as informal as the typical analysis texts, including the use of syncopation for quantifiers.

We shall see how making formal calculation practical for everyday use not only eliminates the style breach (which cynics might consider just a matter of taste), but provides a systematic approach to unify classical and software engineering theories.

1.2 Principle: formal calculation *"ut faciant opus signa"*

Whoever enjoyed physics will recall the excitement when judicious manipulation of formulas yielded results not obtainable by mere intuition. An example is given later, where it also serves to illustrate a paradigm.

The same sense of discovery is seldom imparted by traditional formal logic or the way it is used in developing theories. As explained by Gries [18], although formal logic exists as a separate discipline, its traditional form is drowned in technicalities that make it too cumbersome for practical use, but now calculational variants exist [17].

The rewards of making formal calculation practical for everyday use throughout all areas of mathematics (including logic) are huge.

One advantage is making the symbols do the work, as nicely captured by the maxim "*Ut faciant opus signa*" of the conferences on Mathematics of Program Construction [3]. Here we do not mean only (nor primarily) using software tools, but rather the guidance provided by the shape of the expressions in calculation, and the development of a "parallel intuition" to that effect. This complements the usual "semantic" intuition, especially when exploring areas where the latter is clueless or still in development.

Another advantage is that formal calculation, by shifting emphasis from subject-specific idioms to analogies, common principles and methods, provides unification.

1.3 Realization: Functional Mathematics (*Funmath*)

This lecture presents a formalism spanning a wide application spectrum. By *formalism* we mean a *language* (or notation) together with *formal rules* for symbolic manipulation.

The language [5] is *functional* in the sense that functions are first-class objects and also form the basis for unification. It supports declarative (abstract) as well as operational (implementation) aspects throughout all mathematics relevant to computer and classical engineering, and is free of all defects of common conventions, including those outlined by Lee and Varaiya [25] as discussed later.

The formal rules are *calculational*, supporting the same style of expression manipulation from predicate logic through calculus. Thereby the conceptual and notational unification provided by the language is complemented by unified methodology.

In particular, this enables engineers to formally calculate with predicates and quantifiers with the same ease and algebraic flavor as with derivatives and integrals.

1.4 Overview

The formalism is presented in section 2, which introduces the language, its rationale and its four basic constructs, and in section 3, which gives the general-purpose formal rules, namely those for concrete generic functionals and for functional predicate calculus (quantifiers). Application examples are given in section 4 for Systems Theory, section 5 for Computing Science, and section 6 for common aspects. Some concluding remarks are given in section 7.

The emphasis on Systems Theory in the examples for classical ("continuous") engineering mathematics is justified because the considered abstractions are enlightening to Computing Science as well, and because it shows the relevance of Electrical and Computer Engineering [24] as an emergent unified discipline after an epistemological divorce that has lasted far too long.

2 The formalism, part A: language

The following presentation is derived from the 4-page Funmath LRRL (Language Rationale and Reference Leaflet) issued with a course [8] based on the same approach. The basic principle is defining mathematical concepts as functions whenever it is advantageous; in practice that is the case more often than commonly realized.

2.1 Rationale: the need for defect-free notation

Notation is unimportant if and only if it is well-designed, but becomes crucial as a stumbling block if it is deficient. The criterion is formal calculation: if during calculation one has to be on guard for defects, one cannot let the symbols do the work.

In long-standing areas of mathematics such as algebra and analysis, conventions are largely problem-free, but not entirely. Most important are violations of Leibniz's principle, i.e., that equals may always be replaced by equals. An example is ellipsis: writing dots as in $a_0 + a_1 + \ldots + a_n$. By Leibniz's principle, if $a_i = i^2$ and $n = 7$, this should equal $0 + 1 + \ldots + 49$, where more likely a sum of squares is intended. Well-known (but often neglected) defects, also pointed out in [25], stem from writing function application $f(x)$ when the function f itself is intended, as in $y(t) = x(t) * h(t)$ where $*$ is convolution. This causes incorrect instantiation, e.g., $y(t - \tau) = x(t - \tau) * h(t - \tau)$.

The situation worsens proportionally with the needs for Computing Science. An example in discrete mathematics: for the sum symbol $\sum$ many conventions are mutually inconsistent and calculation rules are rarely given, which even leads to errors in mathematical software as reported for *Mathematica* by Pugh [31]. Poorest are the conventions for logic and sets used in daily practice. A typical defect is abusing the set membership relation $\in$ for binding a dummy. Frequent patterns are $\{x \in X \mid p\}$, as in $\{m \in \mathbb{Z} \mid m < n\}$, and $\{e \mid x \in X\}$, as in $\{n \cdot m \mid m \in \mathbb{Z}\}$, where (in the patterns) p is boolean and e any expression. The ambiguity is revealed by taking $y \in Y$ for p and e.

Defects like these prohibit formal rules and explain why, for such expressions, *syncopation* [34] prevails in the literature, namely using mathematical symbols ($\forall$, $\exists$) just as shorthands for words ("for all", "there exists") rather than as a genuine calculus.

2.2 Funmath language design

We do not patch defects ad hoc, but generate correct forms by orthogonal combination of just 4 constructs, gaining new useful forms of expression for free. The basis is *functional*. A *function* f is fully defined by its *domain* $\mathcal{D}\, f$ and its *mapping* (image definition). Here are the constructs.

Identifier: any symbol or string except colon, filter mark, abstraction dot, parentheses, and a few keywords.

Identifiers are *introduced* by *bindings* $i : X \mathbin{\mathbb{\wedge}} p$, read "$i$ in X satisfying p", where i is the (tuple of) identifier(s), X a set and p a proposition. The *filter* $\mathbin{\mathbb{\wedge}} p$ (or **with** p) is optional, e.g., $n : \mathbb{N}$ and $n : \mathbb{Z} \mathbin{\mathbb{\wedge}} n \geq 0$ are interchangeable. Identifiers from i should not appear in expression X. Shorthand: $i := e$ stands for $i : \iota\, e$. We write $\iota\, e$, not $\{e\}$, for singleton sets, using ι defined by $e' \in \iota\, e \;\equiv\; e' = e$.

Identifiers can be *variables* (in an abstraction) or *constants* (declared by **def** *binding*). Well-established symbols, such as $\mathbb{B}$, $\Rightarrow$, $\mathbb{R}$, $+$, serve as predefined constants.

Application For function f and argument e, the default is $f\,e$; other conventions are specified by dashes in the operator's binding, e.g., $-\!\star\!-$ for infix. For clarity, parentheses are *never* used as operators, but only for parsing. Rules for making them optional are the usual ones. If f is a function-valued function, $f\,x\,y$ stands for $(f\,x)\,y$.

Let $\star$ be infix. *Partial application* is of the form $a\star$ or $\star b$, defined by $(a\star)\,b = a\star b = (\star b)\,a$. *Variadic application* is of the form $a\star b\star c$ etc., and is *always* defined to equal $F\,(a,b,c)$ for a suitably defined *elastic extension* F of $\star$. These two formal styles of application yield more flexibility than the usual informal conventions.

Abstraction The form is $b\,.\,e$, where b is a binding and e an expression (extending after "$.$" as far as parentheses present allow). Intuitively, $v : X \wedge p\,.\,e$ denotes a function whose domain is the set of v in X satisfying p, and mapping v to e. Formally, it is a lambda term [1] extended with typing. The axioms are given in section 3.

Syntactic sugar: one may write $e \mid b$ for $b\,.\,e$ and $v : X \mid p$ for $v : X \wedge p\,.\,v$.

A trivial example: if v does not occur (free) in e, we define $\bullet$ by $X \bullet e = v : X\,.\,e$ to denote *constant functions*. Special cases are the *empty function* $\varepsilon := \emptyset \bullet e$ (any e) and defining $\mapsto$ by $e' \mapsto e = \iota\,e' \bullet e$ for *one-point functions*.

We shall see how abstractions synthesize familiar expressions such as $\sum i : 0\,..\,n\,.\,q^i$ and $\{m : \mathbb{Z} \mid m < n\}$.

Tupling The 1-dimensional form is e, e', e'' (any length), denoting a function with domain axiom $\mathcal{D}\,(e, e', e'') = \{0, 1, 2\}$ and mapping $(e, e', e'')\,0 = e$, $(e, e', e'')\,1 = e'$ etc. The empty tuple is ε and for singleton tuples we define τ with $\tau\,e = 0 \mapsto e$.

Parentheses are *not* part of tupling, and are as optional in (m, n) as in $(m + n)$. Matrices are 2-dimensional tuples.

3 The formalism, part B: rules

The formal calculation rules and gaining fluency with them by numerous exercises is the topic of a full course [8], so here we must be rather terse. Any remark or claim that is not a definition and for which no proof is given can be taken as an exercise.

3.1 Rules for equational and calculational reasoning

Derivation (1) shows the *equational* style: chaining steps by transitivity of "=". Derivation (2) shows generalization to *calculational* reasoning. The general form is

$$\begin{array}{lll} e_0 & R_1\,\langle \text{Justification}_1 \rangle & e_1 \\ & R_2\,\langle \text{Justification}_2 \rangle & e_2\ , \end{array} \tag{3}$$

where the R_i in successive lines are mutually transitive, for instance $=$, $\leq$, etc. in arithmetic, $\equiv$, $\Rightarrow$ etc. in logic. This layout is due to Feyen.

In general (see e.g. Gries [17]), for any theorem p we have the inference rule

$$\textsc{Instantiation:}\ \frac{p}{p[v := e]}\ . \tag{4}$$

We write $d[v := e]$ or $d[^v_e$ for expression d in which every (free) occurrence of variable v is replaced by expression e. Replacement can be multiple, e.g., $(x + y = y + x)[^{x,y}_{3,z+1}$.

For equational reasoning (i.e., using $=$ or $\equiv$ only), the basic rules [17] are reflexivity, symmetry, transitivity and

$$\text{Leibniz's principle: } \frac{e = e'}{d[v := e] = d[v := e']} \ . \tag{5}$$

For instance, $x + 3 \cdot y \quad = \langle x = z^2 \rangle \quad z^2 + 3 \cdot y$. The use of (5) is illustrated by taking $d := v + 3 \cdot y$ and $e := x$ and $e' := z^2$.

3.2 Rules for calculating with propositions and sets

Assume the usual propositional operators $\neg$, $\equiv$, $\Rightarrow$, $\wedge$, $\vee$. For a practical calculus, a much more extensive set of rules is needed than given in classical texts on logic, so we refer to Gries [17]. Note that $\equiv$ is associative, but $\Rightarrow$ is not. We make parentheses in $p \Rightarrow (q \Rightarrow r)$ optional, hence required in $(p \Rightarrow q) \Rightarrow r$. We embed binary algebra in arithmetic [4, 5] with logic constants are 0 and 1 (some may prefer False and True).

Leibniz's principle can be rewritten $e = e' \Rightarrow d[^v_e = d[^v_{e'}$.

For sets, the basic operator is $\in$. The rules are derived ones, e.g., defining $\cap$ by $x \in X \cap Y \ \equiv \ x \in X \wedge x \in Y$ and $\times$ by $(x, y) \in X \times Y \ \equiv \ x \in X \wedge y \in Y$. After defining $\{—\}$, we shall be able to prove $y \in \{x : X \mid p\} \ \equiv \ y \in X \wedge p[^x_y$.

Set equality is defined via *Leibniz's principle*, written as an implication: $X = Y \Rightarrow (x \in X \ \equiv \ x \in Y)$ and the converse, *extensionality*, written here as an inference rule: from $x \in X \ \equiv \ x \in Y$, infer $X = Y$, with x a new variable. This rule is *strict*, i.e., the premiss must be a theorem.

3.3 Rules for functions and generic functionals

We omit the design decisions, to be found in [6] and [9]. In what follows, f and g are any functions, P any predicate ($\mathbb{B}$-valued function, $\mathbb{B} := \{0, 1\}$), X any set, e arbitrary.

3.3.1 Function equality and abstraction

Equality is defined via *Leibniz's principle*, taking domains into account:

$$\text{Leibniz for functions: } f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (e \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, e = g\, e) \tag{6}$$

and its converse, expressed as a strict inference rule: with new v,

$$\text{Function extensionality: } \frac{p \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (v \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, v = g\, v)}{p \Rightarrow f = g} \tag{7}$$

(Function) *abstraction* encapsulates substitution. The formal axioms are

$$\text{Domain axiom: } \quad d \in \mathcal{D}\,(v : X \mathrel{|\!\wedge} p \,.\, e) \quad \equiv \quad d \in X \wedge p[^v_d \tag{8}$$

$$\text{Mapping axiom: } \quad d \in \mathcal{D}\,(v : X \mathrel{|\!\wedge} p \,.\, e) \quad \Rightarrow \quad (v : X \mathrel{|\!\wedge} p \,.\, e)\, d = e[^v_d \tag{9}$$

Equality is characterized via function equality (exercise).

Next, we introduce *(concrete) generic functionals*. The qualification *concrete* emphasizes a distinction from category theory.

A first batch does not require predicate calculus, but is used in deriving the rules of predicate calculus in point-free form. A later batch relies on predicate calculus.

3.3.2 Generic functionals

The purpose is (a) removing restrictions in common functionals from mathematics, (b) making often-used implicit functionals from systems theory explicit. The idea is defining the result domain to avoid out-of-domain applications in the image definition.

Case (a) is illustrated by composition $f \circ g$. Common definitions require $\mathcal{R}\, g \subseteq \mathcal{D}\, f$, and then $\mathcal{D}\,(f \circ g) = \mathcal{D}\, g$. Removing the restriction, we define $f \circ g$ for *any* functions:

$$f \circ g \;=\; x : \mathcal{D}\, g \mathbin{\&} x \in \mathcal{D}\, f \,.\, f\,(g\, x) \;. \tag{10}$$

Observation: $x \in \mathcal{D}\,(f \circ g) \;\equiv\; x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f$ by the abstraction axiom, hence $\mathcal{D}\,(f \circ g) = \{x : \mathcal{D}\, g \mid g\, x \in \mathcal{D}\, f\}$. The generalization is *conservative*: it coincides with the common convention in case the restriction $\mathcal{R}\, g \subseteq \mathcal{D}\, f$ is satisfied. Keeping generalizations conservative is a secondary design criterion for generic functionals.

Case (b) is illustrated by the usual implicit generalization of arithmetic functions to signals, traditionally written $(s + s')(t) = s(t) + s'(t)$. We generalize this as follows.

(Duplex) direct extension ($\widehat{\ }$): for any functions $\star$ (infix), f, g,

$$f \mathbin{\widehat{\star}} g \;=\; x : \mathcal{D}\, f \cap \mathcal{D}\, g \mathbin{\&} (f\, x, g\, x) \in \mathcal{D}\,(\star) \,.\, f\, x \star g\, x \;. \tag{11}$$

Often we need *half direct extension* ($\overleftarrow{\ }$ and $\overrightarrow{\ }$): for function f, any e,

$$f \mathbin{\overleftarrow{\star}} e = f \mathbin{\widehat{\star}} (\mathcal{D}\, f \bullet e) \quad \text{and} \quad e \mathbin{\overrightarrow{\star}} f = (\mathcal{D}\, f \bullet e) \mathbin{\widehat{\star}} f \;. \tag{12}$$

recalling for easy reference the operator $\bullet$ for defining *constant functions*:

$$X \bullet e \;=\; v : X \,.\, e \qquad (v \text{ not free in } e) \tag{13}$$

Simplex direct extension ($\overline{\ }$) is defined by $\overline{f}\, g = f \circ g$.

Function merge ($\cup$) is defined in 2 parts to fit the line:

$$\begin{aligned} x \in \mathcal{D}\,(f \cup g) \;&\equiv\; x \in \mathcal{D}\, f \cup \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \\ x \in \mathcal{D}\,(f \cup g) \;&\Rightarrow\; (f \cup g)\, x = (x \in \mathcal{D}\, f)\, ?\, f\, x \mathbin{\dagger} g\, x \end{aligned} \tag{14}$$

Here we used a *conditional*. The general form is $b\, ?\, e_1 \mathbin{\dagger} e_0$, read "if b then e_1 else e_0". Its rule is $(b\, ?\, e_1 \mathbin{\dagger} e_0) = e_b$; equivalently: $(b\, ?\, e_1 \mathbin{\dagger} e_0) = (e_0, e_1)\, b$.

Filtering ($\downarrow$) introduces/eliminates arguments:

$$f \downarrow P \;=\; x : \mathcal{D}\, f \cap \mathcal{D}\, P \mathbin{\&} P\, x \,.\, f\, x \;. \tag{15}$$

A particularization is the more familiar *restriction* ($\rceil$) defined by $f \rceil X = f \downarrow (X \bullet 1)$. We extend $\downarrow$ to sets by $x \in (X \downarrow P) \;\equiv\; x \in X \cap \mathcal{D}\, P \wedge P\, x$.

Writing a_b for $a \downarrow b$ and using partial application, function and set filtering yields formal rules for useful shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$.

Relational generic functionals Subfunction ($\subseteq$) with $f \subseteq g \;\equiv\; f = g \rceil \mathcal{D}\, f$ and compatibility ($\copyright$) with $f \copyright g \;\equiv\; f \rceil \mathcal{D}\, g = g \rceil \mathcal{D}\, f$. Note $f = g \;\equiv\; \mathcal{D}\, f = \mathcal{D}\, g \wedge f \copyright g$.

For many other generic functionals and their elastic extensions, we refer to [9].

A very important use of generic functionals is supporting the *point-free* style, i.e., without referring to domain points. The elegant algebraic flavor is illustrated next.

3.4 Rules for predicate calculus and quantifiers

3.4.1 Axioms and forms of expression

The *quantifiers* $\forall$, $\exists$ are predicates over predicates: for any predicate P,

$$\forall P \;\equiv\; P = \mathcal{D}\,P^\bullet 1 \quad\text{and}\quad \exists P \;\equiv\; P \neq \mathcal{D}\,P^\bullet 0 \;. \tag{16}$$

Let p and q be propositions, then p, q is a predicate and $\forall\,(p,q) \equiv p \wedge q$. So $\forall$ is an elastic extension of $\wedge$ and we define variadic application by $p \wedge q \wedge r \equiv \forall\,(p,q,r)$ etc.

If P is an abstraction $v : X \,.\, p$, clearly $\forall P \equiv \forall v : X \,.\, p$. For every algebraic law, most elegantly stated in point-free form, this yields a familiar-looking pointwise form.

3.4.2 Derived rules

All laws follow from (16) and function equality. A collection sufficient for practical use is derived in [8]. Here we only give some examples, starting with a formula for $f = g$:

$$f = g \;\equiv\; \mathcal{D}\,f = \mathcal{D}\,g \;\wedge\; \forall x : \mathcal{D}\,f \cap \mathcal{D}\,g \,.\, f\,x = g\,x \;. \tag{17}$$

Another example is *duality* (generalizing De Morgan law)

$$\overline{\neg} \forall P \equiv \exists\,(\overline{\neg} P) \qquad \neg\,(\forall v : X \,.\, p) \equiv \exists v : X \,.\, \neg p \;. \tag{18}$$

Here are the main distributivity laws. All have duals (not stated here):

Name of the rule	Point-free form	Letting $P := v : X \,.\, p$ with $v \notin \varphi\, q$
Distributivity $\vee/\forall$	$q \vee \forall P \equiv \forall\,(q \mathbin{\vec{\vee}} P)$	$q \vee \forall\,(v : X \,.\, p) \equiv \forall\,(v : X \,.\, q \vee p)$
L(eft)-distrib. $\Rightarrow/\forall$	$q \Rightarrow \forall P \equiv \forall\,(q \mathbin{\vec{\Rightarrow}} P)$	$q \Rightarrow \forall\,(v : X \,.\, p) \equiv \forall\,(v : X \,.\, q \Rightarrow p)$
R(ight)-distr. $\Rightarrow/\exists$	$\exists P \Rightarrow q \equiv \forall\,(P \mathbin{\overleftarrow{\Rightarrow}} q)$	$\exists\,(v : X \,.\, p) \Rightarrow q \equiv \forall\,(v : X \,.\, p \Rightarrow q)$
P(seudo)-dist. $\wedge/\forall$	$q \wedge \forall P \equiv \forall\,(q \mathbin{\vec{\wedge}} P)$	$q \wedge \forall\,(v : X \,.\, p) \equiv \forall\,(v : X \,.\, q \wedge p)$

P(seudo)-distributivity $\wedge/\forall$ as stated (to fit the table) assumes $\mathcal{D}\,P \neq \emptyset$; the general form is $\mathcal{D}\,P = \emptyset \vee (q \wedge \forall P) \equiv \forall\,(q \mathbin{\vec{\wedge}} P)$. Aside: $\varphi\, e$ is the set of free variables in e.

Here are a few important additional rules (again leaving duals as an exercise).

Name	Point-free form	Letting $P := v : X \,.\, p$ with $v \notin \varphi\, q$
Distrib. $\forall/\wedge$	$\forall\,(P \mathbin{\widehat{\wedge}} Q) \equiv \forall P \wedge \forall Q$	$\forall\,(v : X \,.\, p \wedge q) \equiv \forall\,(v : X \,.\, p) \wedge \forall\,(v : X \,.\, q)$
One-pt. rule	$\forall P_{=e} \equiv e \in \mathcal{D}\,P \Rightarrow P\,e$	$\forall\,(v : X \,.\, v = e \Rightarrow p) \equiv e \in X \Rightarrow p[^v_e]$
Trading $\forall$	$\forall P_Q \equiv \forall\,(Q \mathbin{\widehat{\Rightarrow}} P)$	$\forall\,(v : X \wedge q \,.\, p) \equiv \forall\,(v : X \,.\, q \Rightarrow p)$
Transp./Swap	$\forall\,(\forall \circ R) \equiv \forall\,(\forall \circ R^\mathsf{T})$	$\forall\,(v : X \,.\, \forall\, w : Y \,.\, p) \equiv \forall\,(w : Y \,.\, \forall\, v : X \,.\, p)$

Distributivity $\forall/\wedge$ assumes $\mathcal{D}\,P = \mathcal{D}\,Q$, otherwise only $\forall P \wedge \forall Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q)$. For the last line, $R : X \to Y \to \mathbb{B}$ and transposition $^\mathsf{T}$ satisfies $(v : X \,.\, w : Y \,.\, e)^\mathsf{T} = w : Y \,.\, v : X \,.\, e$.

Sometimes the following rules are useful (metatheorems matching axioms in logic):

INSTANTIATION: $\quad \forall P \Rightarrow e \in \mathcal{D}\,P \Rightarrow P\,e$

GENERALIZATION: $\quad p \Rightarrow v \in \mathcal{D}\,P \Rightarrow P\,v \;\vdash\; p \Rightarrow \forall P \qquad$ (new v)

Rules not shown in this overview but invoked later will be clear from the nomenclature used. Observe that most rules are *equational*, involving "$\equiv$" only. The derivations (left as exercises) may initially require proving ($\Rightarrow$) and ($\Leftarrow$) separately and invoking *mutual implication* [17], but the need for doing so diminishes as rules accumulate.

An example is the following proof of (17). We show ($\Rightarrow$), since ($\Leftarrow$) is similar.

$$\begin{array}{lll}
f = g & \Rightarrow \langle \text{Leibniz (6)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \\
& \equiv \langle p \equiv p = 1 \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow (f\, x = g\, x) = 1) \\
& \equiv \langle \text{Def. } \widehat{=} \text{ (11)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, (f \mathrel{\widehat{=}} g) \Rightarrow (f \mathrel{\widehat{=}} g)\, x = 1 \\
& \equiv \langle \text{Def. } \bullet \text{ (13)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, (f \mathrel{\widehat{=}} g) \Rightarrow (f \mathrel{\widehat{=}} g)\, x = (\mathcal{D}\, (f \mathrel{\widehat{=}} g) \bullet 1)\, x \\
& \Rightarrow \langle \text{Extns. (7)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (f \mathrel{\widehat{=}} g) = \mathcal{D}\, (f \mathrel{\widehat{=}} g) \bullet 1 \\
& \equiv \langle \text{Def. } \forall \text{ (16)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge \forall\, (f \mathrel{\widehat{=}} g)
\end{array}$$

3.5 More on functions and generic functionals

3.5.1 Wrapping up the rule package for functions

Function range We define the range operator $\mathcal{R}$ by

$$e \in \mathcal{R}\, f \;\equiv\; \exists\, x : \mathcal{D}\, f \,.\, f\, x = e \;. \tag{19}$$

Theorem: composition rules $\forall\, P \Rightarrow \forall\, (P \circ f)$ and $\mathcal{D}\, P \subseteq \mathcal{R}\, f \Rightarrow (\forall\, (P \circ f) \;\equiv\; \forall\, P)$. The pointwise form yields $\forall\, (y : \mathcal{R}\, f \,.\, p) \;\equiv\; \forall\, (x : \mathcal{D}\, f \,.\, p[^{y}_{f\, x}])$, called "dummy change".

The familiar *function arrow* ($\rightarrow$) is defined by $f \in X \rightarrow Y \;\equiv\; \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$. Similarly, the *partial arrow* ($\nrightarrow$) is defined by $f \in X \nrightarrow Y \;\equiv\; \mathcal{D}\, f \subseteq X \wedge \mathcal{R}\, f \subseteq Y$.

Set comprehension We define $\{—\}$ as *fully interchangeable* with $\mathcal{R}$. This yields defect-free set notation: expressions like $\{2, 3, 5\}$ and $\mathit{Even} = \{2 \cdot m \mid m : \mathbb{Z}\}$ have familiar form and meaning, and all desired calculation rules follow from predicate calculus via (19). In particular, we can prove $e \in \{v : X \mid p\} \;\equiv\; e \in X \wedge p[^{v}_{e}$ (exercise).

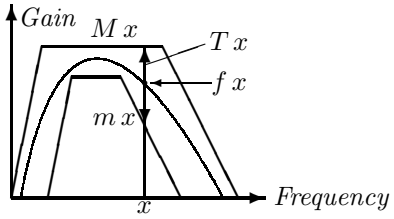
Function inverse The generic inverse is based on the *bijectivity domain and range*.

$$\left.\begin{array}{l}
\mathsf{Bdom}\, f \;=\; \{x : \mathcal{D}\, f \mid \forall\, y : \mathcal{D}\, f \,.\, f\, x = f\, y \Rightarrow x = y\} \\
\mathsf{Bran}\, f \;=\; \{f\, x \mid x : \mathsf{Bdom}\, f\} \\
f^{-} \in \mathsf{Bran}\, f \rightarrow \mathsf{Bdom}\, f \quad \text{and} \quad \forall\, x : \mathsf{Bdom}\, f \,.\, f^{-}\, (f\, x) = x
\end{array}\right\} \tag{20}$$

3.5.2 Designing a generic functional from the *function tolerance* paradigm

We design a functional motivated by analog electronics but having wide applications in computing. The presentation in 3 steps: paradigm, generalization, particularization is taken from [9], where other unifying abstractions are developed similarly.

(i) The *starting point* is formalizing a convention for specifying a radio frequency filter characteristic f [12] within a given tolerance, e.g., $m\, x \leq f\, x \leq M\, x$.



To this effect, we let T be an interval-valued function, e.g., $T\, x = [m\, x, M\, x]$, and say that function f *meets tolerance* T iff $\mathcal{D}\, f = \mathcal{D}\, T$ and, for all x in the domain, $f\, x \in T\, x$.

(ii) The (small) *generalization step* is allowing *any* (not just "dense") sets for $\mathcal{D}\,T$. This suggests defining an operator $\times$ such that, if T is a set-valued function,

$$f \in \times\, T \quad \equiv \quad \mathcal{D}\,f = \mathcal{D}\,T \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,T \,.\, f\,x \in T\,x \tag{21}$$

Using (17), calculation yields $f = g \;\equiv\; f \in \times\,(\iota \circ g)$, so the tolerance can be *tight*.

(iii) *Particularization step:* Calculating (21) for $T := A, B$ (a pair of sets) yields

$$f \in \times\,(A, B) \quad \equiv \quad \mathcal{D}\,f = \mathbb{B} \wedge f\,0 \in A \wedge f\,1 \in B \;\;.$$

So $\times\,(A, B) = A \times B$, the usual product set (tuples being functions). This explains the symbol $\times$ and the name *generalized functional Cartesian* (or *funcart*) *product*. Also, $\times\,(A \bullet B) = A \to B$, so $\times$ covers all "ordinary" function types as well.

As expected, $\times$ defines variadic shorthand for $\times$, as in $A \times B \times C = \times\,(A, B, C)$. Applied to abstractions, as in $\times\, a : A \,.\, B\,a$, it covers so-called *dependent types* or *product types* [35], in the literature often denoted by ad hoc abstractions like $\prod_{a \in A} B\,a$. We also introduce the suggestive shorthand $A \ni a \to B\,a$ for $\times\, a : A \,.\, B\,a$, which is especially convenient in chained dependencies, e.g., $A \ni a \to B\,a \ni b \to C\,a\,b$.

(iv) *Additional properties.* In contrast with ad hoc abstractions $\prod_{a \in A} B\,a$, the operator $\times$ is a functional and has many useful algebraic properties. Noteworthy is the inverse. By the axiom of choice, $\times\, T \neq \emptyset \;\equiv\; \forall\, x : \mathcal{D}\,T \,.\, T \neq \emptyset$. This also characterizes the bijectivity domain of $\times$ and, if $\times\, T \neq \emptyset$, then $\times^{-}\,(\times\, T) = T$. For the usual cartesian product this implies that, if $A \neq \emptyset$ and $B \neq \emptyset$, then $\times^{-}\,(A \times B) = A, B$.

An explicit image definition for $\times^{-}$ is $\times^{-}\, S = x : \bigcap (f : S \,.\, \mathcal{D}\,f) \,.\, \{f\,x \mid f : S\}$ for any nonempty S in the range of $\times$.

4 Examples I: Systems Theory

4.1 Analysis: calculation replacing syncopation

Proofs traditionally rendered tedious by syncopation [34] are cleaner formally.

Adjacency This example is an exercise from [22]. Since predicates (of type $\mathbb{R} \to \mathbb{B}$) yield more elegant formulations than sets (of type $\mathcal{P}\,\mathbb{R}$), we define the predicate transformer $\mathsf{ad} : (\mathbb{R} \to \mathbb{B}) \to (\mathbb{R} \to \mathbb{B})$ and the predicates open and closed, of type $(\mathbb{R} \to \mathbb{B}) \to \mathbb{B}$:

$$\begin{aligned}
\mathsf{ad}\,P\,v \;&\equiv\; \forall\, \epsilon : \mathbb{R}_{>0} \,.\, \exists\, x : \mathbb{R}_P \,.\, |x - v| < \epsilon \\
\mathsf{open}\,P \;&\equiv\; \forall\, v : \mathbb{R}_P \,.\, \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, |x - v| < \epsilon \Rightarrow P\,x \\
\mathsf{closed}\,P \;&\equiv\; \mathsf{open}\,(\overline{\neg}\,P)
\end{aligned}$$

The exercise from [22] is proving the *closure property* $\mathsf{closed}\,P \;\equiv\; \mathsf{ad}\,P = P$. We proceed by calculation, assuming the lemma $P\,v \Rightarrow \mathsf{ad}\,P\,v$ (easy to prove), is

$$\begin{array}{lcll}
\mathsf{closed}\,P & \equiv & \langle \text{Def. } \mathsf{closed} \rangle & \mathsf{open}\,(\overline{\neg}\,P) \\
& \equiv & \langle \text{Def. } \mathsf{open} \rangle & \forall\, v : \mathbb{R}_{\overline{\neg}\,P} \,.\, \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, |x - v| < \epsilon \Rightarrow \overline{\neg}\,P\,x \\
& \equiv & \langle \text{Contrapos.} \rangle & \forall\, v : \mathbb{R}_{\overline{\neg}\,P} \,.\, \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, P\,x \Rightarrow \neg\,(|x - v| < \epsilon) \\
& \equiv & \langle \text{Trading } \forall \rangle & \forall\, v : \mathbb{R} \,.\, \overline{\neg}\,P\,v \Rightarrow \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R}_P \,.\, \neg\,(|x - v| < \epsilon) \\
& \equiv & \langle \text{Contrapos.} \rangle & \forall\, v : \mathbb{R} \,.\, \overline{\neg} \exists\, (\epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R}_P \,.\, \neg\,(|x - v| < \epsilon) \Rightarrow P\,v \\
& \equiv & \langle \text{Duality} \rangle & \forall\, v : \mathbb{R} \,.\, \forall\, (\epsilon : \mathbb{R}_{>0} \,.\, \exists\, x : \mathbb{R}_P \,.\, |x - v| < \epsilon) \Rightarrow P\,v \\
& \equiv & \langle \text{Def. } \mathsf{ad} \rangle & \forall\, v : \mathbb{R} \,.\, \mathsf{ad}\,P\,v \Rightarrow P\,v \\
& \equiv & \langle \text{Lemma} \rangle & \forall\, v : \mathbb{R} \,.\, \mathsf{ad}\,P\,v \;\equiv\; P\,v \quad .
\end{array}$$

The set equivalent Ad and its relation to open and closed sets are left as an exercise.

Limits Here we only provide an outline; the calculations are similar but provide a good opportunity for practice. All details can be found in [8].

We define a relation islim_f (parametrized by any function $f : \mathbb{R} \nrightarrow \mathbb{R}$) between $\mathbb{R}$ and the set of points adherent to $\mathcal{D}\, f$, that is: $\mathsf{islim}_f \in \mathbb{R} \times \mathsf{Ad}\,(\mathcal{D}\, f) \to \mathbb{B}$ with

$$L \;\mathsf{islim}_f\, a \quad \equiv \quad \forall\, \epsilon : \mathbb{R}_{>0}\,.\, \exists\, \delta : \mathbb{R}_{>0}\,.\, \forall\, x : \mathcal{D}\, f\,.\, |x - a| < \delta \Rightarrow |f\, x - L| < \epsilon. \tag{22}$$

The affix convention is chosen such that $L\,\mathsf{islim}_f\, a$ is read "L is a limit for f at a".

THEOREM: for any function $f : \mathbb{R} \nrightarrow \mathbb{R}$, subset S of $\mathcal{D}\, f$ and a adherent to S,

$$\text{(i)} \qquad \exists\,(L : \mathbb{R}\,.\, L\,\mathsf{islim}_f\, a) \Rightarrow \exists\,(L : \mathbb{R}\,.\, L\,\mathsf{islim}_{f\restriction S}\, a)\ , \tag{23}$$

$$\text{(ii)} \qquad \forall\, L : \mathbb{R}\,.\, \forall\, M : \mathbb{R}\,.\, L\,\mathsf{islim}_f\, a \wedge M\,\mathsf{islim}_{f\restriction S}\, a \Rightarrow L = M\ . \tag{24}$$

We conclude by defining a functional

$$\begin{array}{l} \mathbf{def}\ \mathsf{lim} : (\mathbb{R} \nrightarrow \mathbb{R}) \ni f \to \{a : \mathsf{Ad}\,(\mathcal{D}\, f) \mid \exists\, b : \mathbb{R}\,.\, b\,\mathsf{islim}_f\, a\} \to \mathbb{R} \\ \mathbf{with}\ \forall\, f : \mathcal{D}\,\mathsf{lim}\,.\, \forall\, a : \mathcal{D}\,(\mathsf{lim}\, f)\,.\, (\mathsf{lim}\, f\, a)\,\mathsf{islim}_f\, a\ . \end{array} \tag{25}$$

Well-definedness follows from function comprehension and uniqueness (24).

Note that lim is a functional, not an ad hoc abstractor, and supports point-free expressions like $\mathsf{lim}\, f\, a$ (versus $\lim_{x \to a} f\, x$). Second, domain modulation covers left, right and two-sided limits, e.g., given $\mathbf{def}\ f : \mathbb{R} \to \mathbb{R}\ \mathbf{with}\ f\, x = (x \geq 0)\,?\,0 \nmid x + 1$, then $\mathsf{lim}\, f_{<0}\, 0 = 1$ and $\mathsf{lim}\, f_{\geq 0}\, 0 = 0$, whereas $0 \notin \mathcal{D}\,(\mathsf{lim} f_{\leq 0})$.

No separate concepts or notations are needed. Conventional notations can be synthesized from lim by macro definitions such that, for instance, if e is a real expression, $\lim\limits_{x \to a} e$ stands for $\mathsf{lim}\,(x : \mathbb{R}\,.\, e)\, a$ and $\lim\limits_{\substack{x \to a \\ <}} e$ stands for $\mathsf{lim}\,(x : \mathbb{R}_{<a}\,.\, e)\, a$ and so on.

Derivatives (outline) We define the *Newton quotient functional*

$$\mathbf{def}\ Q : FD \ni f \to \mathcal{D}\, f \ni x \to \mathcal{D}\, f \setminus \iota\, x \to \mathbb{R}\ \mathbf{with}\ Q\, f\, x\, y = \frac{f\, y - f\, x}{y - x}\ ,$$

where FD is the set of real-valued functions whose domain is an *interval* [22] of more than one point. With this auxiliary function, we define the *derivation* operator with image definition $\mathrm{D}\, f\, x = \mathsf{lim}\,(Q\, f\, x)\, x$, the type definition being left as an exercise.

4.2 An example about transform methods

We show how formally correct use of functionals, in particular replacing common defective notations like $\mathcal{F}\,\{f(t)\}$ by $\mathcal{F}\, f\, \omega$, enables formal calculation. In

$$\begin{array}{rcl} \mathcal{F}\, f\, \omega & = & \int_{-\infty}^{+\infty} e^{-j \cdot \omega \cdot t} \cdot f\, t \cdot \mathrm{d}\, t \\ \mathcal{F}'\, g\, t & = & \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} e^{j \cdot \omega \cdot t} \cdot g\, \omega \cdot \mathrm{d}\, \omega \end{array}$$

bindings are clear and unambiguous. The example formalizes Laplace transforms via Fourier transforms. We assume some familiarity with the usual informal treatments.

Given $\ell\_ : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ with $\ell_\sigma\, t = (t < 0)\,?\,0 \nmid e^{-\sigma \cdot t}$, we define the Laplace-transform $\mathcal{L}\, f$ of a function f by:

$$\mathcal{L}\, f\,(\sigma + j \cdot \omega) \quad = \quad \mathcal{F}\ (\ell_\sigma \mathbin{\widehat{\ }} f)\, \omega \tag{26}$$

for real σ and ω, with σ such that $\ell_\sigma \mathbin{\hat{\cdot}} f$ has a Fourier transform. With $s := \sigma + j \cdot \omega$ we obtain $\mathcal{L}\, f\, s = \int_0^{+\infty} f\, t \cdot e^{-s \cdot t} \cdot \mathrm{d}\, t$.

The converse $\mathcal{L}'$ is specified by $\mathcal{L}'\,(\mathcal{L}\, f)\, t = f\, t$ for all $t \geq 0$ (weakened where $\ell_\sigma \mathbin{\hat{\cdot}} f$ is discontinous). For such t,

$$\begin{array}{rcll}
\mathcal{L}'\,(\mathcal{L}\, f)\, t & = & \langle \text{Specification} \rangle & f\, t \\
& = & \langle e^{\sigma \cdot t} \cdot \ell_\sigma\, t = 1 \rangle & e^{\sigma \cdot t} \cdot \ell_\sigma\, t \cdot f\, t \\
& = & \langle \text{Definition } \hat{} \rangle & e^{\sigma \cdot t} \cdot (\ell_\sigma \mathbin{\hat{\cdot}} f)\, t \\
& = & \langle \text{Weakening} \rangle & e^{\sigma \cdot t} \cdot \mathcal{F}'\,(\mathcal{F}\ (\ell_\sigma \mathbin{\hat{\cdot}} f))\, t \\
& = & \langle \text{Definition } \mathcal{F}' \rangle & e^{\sigma \cdot t} \cdot \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{F}\ (\ell_\sigma \mathbin{\hat{\cdot}} f)\, \omega \cdot e^{j \cdot \omega \cdot t} \cdot \mathrm{d}\, \omega \\
& = & \langle \text{Definition } \mathcal{L} \rangle & e^{\sigma \cdot t} \cdot \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{L}\, f\, (\sigma + j \cdot \omega) \cdot e^{j \cdot \omega \cdot t} \cdot \mathrm{d}\, \omega \\
& = & \langle \text{Const. factor} \rangle & \frac{1}{2 \cdot \pi} \cdot \int_{-\infty}^{+\infty} \mathcal{L}\, f\, (\sigma + j \cdot \omega) \cdot e^{(\sigma + j \cdot \omega) \cdot t} \cdot \mathrm{d}\, \omega \\
& = & \langle s := \sigma + j \cdot \omega \rangle & \frac{1}{2 \cdot \pi \cdot j} \cdot \int_{\sigma - j \cdot \infty}^{\sigma + j \cdot \infty} \mathcal{L}\, f\, s \cdot e^{s \cdot t} \cdot \mathrm{d}\, s
\end{array}$$

4.3 Characterization and properties of systems

General A *signal* over a value space A is a function of type $\mathcal{S}_A$ with $\mathcal{S}_A = \mathbb{T} \to A$ for some time domain $\mathbb{T}$. A *system* is a function $s : \mathcal{S}_A \to \mathcal{S}_B$. The response of s to input signal $x : \mathcal{S}_A$ at time $t : \mathbb{T}$ is $s\, x\, t$, read $(s\, x)\, t$.

Characteristics Let $s : \mathcal{S}_A \to \mathcal{S}_B$.

Then s is *memoryless* iff $\exists\, f_\_ : \mathbb{T} \to A \to B \,.\, \forall\, x : \mathcal{S}_A \,.\, \forall\, t : \mathbb{T} \,.\, s\, x\, t = f_t\, (x\, t)$.

Let $\mathbb{T}$ be additive, and the *shift* function $\sigma_\_$ be defined by $\sigma_\tau\, x\, t = x\, (t + \tau)$ for any t and τ in $\mathbb{T}$ and any signal x. Then system s is *time-invariant* iff $\forall\, \tau : \mathbb{T} \,.\, s \circ \sigma_\tau = \sigma_\tau \circ s$.

A system $s : \mathcal{S}_\mathbb{R} \to \mathcal{S}_\mathbb{R}$ is *linear* iff $\forall\, (x, y) : \mathcal{S}_\mathbb{R}^2 \,.\, \forall\, (a, b) : \mathbb{R}^2 \,.\, s\, (a \mathbin{\vec{\cdot}} x \mathbin{\hat{+}} b \mathbin{\vec{\cdot}} y) = a \mathbin{\vec{\cdot}} s\, x \mathbin{\hat{+}} b \mathbin{\vec{\cdot}} s\, y$. Equivalently, extending s to $\mathcal{S}_\mathbb{C} \to \mathcal{S}_\mathbb{C}$ in the evident way, s is *linear* iff $\forall\, z : \mathcal{S}_\mathbb{C} \,.\, \forall\, c : \mathbb{C} \,.\, s\, (c \mathbin{\vec{\cdot}} z) = c \mathbin{\vec{\cdot}} s\, z$.

A system is LTI iff it is both linear and time-invariant.

Response of LTI systems Define the parametrized exponential $\mathsf{E}_\_ : \mathbb{C} \to \mathbb{T} \to \mathbb{C}$ by $\mathsf{E}_c\, t = e^{c \cdot t}$. Then we have:

THEOREM: if s is LTI then $s\, \mathsf{E}_c = s\, \mathsf{E}_c\, 0 \mathbin{\vec{\cdot}} \mathsf{E}_c$.

Proof: we calculate $s\, \mathsf{E}_c\, (t + \tau)$ to exploit all properties.

$$\begin{array}{rcll}
s\, \mathsf{E}_c\, (t + \tau) & = & \langle \text{Definition } \sigma \rangle & \sigma_\tau\, (s\, \mathsf{E}_c)\, t \\
& = & \langle \text{Time inv. } s \rangle & s\, (\sigma_\tau\, \mathsf{E}_c)\, t \\
& = & \langle \text{Property } \mathsf{E}_c \rangle & s\, (\mathsf{E}_c\, \tau \mathbin{\vec{\cdot}} \mathsf{E}_c)\, t \\
& = & \langle \text{Linearity } s \rangle & (\mathsf{E}_c\, \tau \mathbin{\vec{\cdot}} s\, \mathsf{E}_c)\, t \\
& = & \langle \text{Defintion } \vec{} \rangle & \mathsf{E}_c\, \tau \cdot s\, \mathsf{E}_c\, t
\end{array}$$

Substituting $t := 0$ yields $s\, \mathsf{E}_c\, \tau = s\, \mathsf{E}_c\, 0 \cdot \mathsf{E}_c\, \tau$ or, using $\vec{}$, $s\, \mathsf{E}_c\, \tau = (s\, \mathsf{E}_c\, 0 \mathbin{\vec{\cdot}} \mathsf{E}_c)\, \tau$, so $s\, \mathsf{E}_c = s\, E_c\, 0 \mathbin{\vec{\cdot}} \mathsf{E}_c$ by function equality. The $\langle \text{Property } \mathsf{E}_c \rangle$ is $\sigma_\tau\, \mathsf{E}_c = \mathsf{E}_c\, \tau \mathbin{\vec{\cdot}} \mathsf{E}_c$ (easy). Note that this proof uses only the essential hypotheses.

5 Examples II: Computing Science

5.1 From data structures to query languages

Records as in PASCAL [21] are expressed by $\times$ as *functions* whose domain is a set of field labels (an *enumeration type*). Example: with field names `name` and `age`,

$$Person := \times\,(\mathtt{name} \mapsto \mathbb{A}^* \uplus \mathtt{age} \mapsto \mathbb{N})$$

defines a function type such that $person : Person$ satisfies $person\ \mathtt{name} \in \mathbb{A}^*$ and $person\ \mathtt{age} \in \mathbb{N}$.

Obviously, by defining $\mathsf{record}\, F = \times\,(\bigcup F)$ ($\bigcup$: elastic extension of $\uplus$), one can also write $Person := \mathsf{record}\,(\mathtt{name} \mapsto \mathbb{A}^*, \mathtt{age} \mapsto \mathbb{N})$.

Trees are functions whose domains are *branching structures*, i.e., sets of sequences describing the path from the root to a leaf in the obvious way (for any branch labeling). Other structures are covered similarly

Relational databases The expression

$$\mathsf{record}\,(\mathsf{code} \mapsto \mathit{Code}, \mathsf{name} \mapsto \mathbb{A}^*, \mathsf{inst} \mapsto \mathit{Staff}, \mathsf{prrq} \mapsto \mathit{Code}^*)$$

specifies the type of tables of the form

Code	Name	Instructor	Prerequisites
CS100	Elements of logic	R. Barns	none
MA115	Basic Probability	K. Jason	MA100
CS300	Formal Methods	R. Barns	CS100, EE150
...	...	...	

Generic functionals subsume all usual query-operators:

For the *selection*-operator (σ): $\sigma\,(S, P) = S \downarrow P$.

For *projection* (π): $\pi\,(S, F) = \{r \rceil F \mid r : S\}$.

For the *join*-operator ($\bowtie$): $S \bowtie T = S \otimes T$.

Here $\otimes$ is the generic *function type merge* operator, defined as in [9] by $S \otimes T = \{s \uplus t \mid (s, t) : S \times T \wedge s \,©\, t\}$. Note that $\otimes$ is associative, although $\uplus$ is not (exercise).

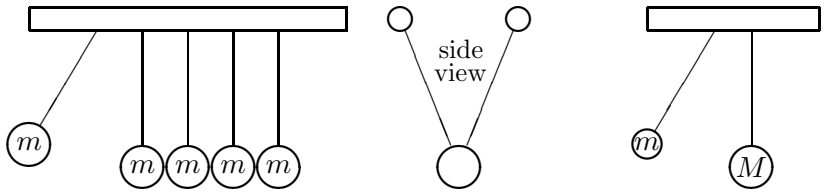
5.2 Deriving theories of programming

We show how the functional predicate calculus unifies the methodology for mathematical analysis (the example about adjacency) and theories of programming.

Starting with analogy from physics, we hope to convey a similar sense of discovery by deriving well-known axiomatic semantics —which in its customary form looks like a rabbit out of a hat— from more basic principles, namely program equations.

5.2.1 An example from physics: state changes in discretized time

Newton's Cradle is a desk adornment consisting of steel balls hanging in a straight row from a frame. It is meant to illustrate the mechanics of collision. Two variants are shown below. We consider the rightmost one with two balls of mass m and M.



With obvious conventions, the state s is a pair of velocities: $s = v, V$. The states $\backprime s$ before and s' just after collision are related by conservation of momentum and, assuming lossless collisions, conservation of energy. Combined into one relation R,

$$\begin{aligned} R\,(\backprime s, s') \quad &\equiv \quad m \cdot \backprime v + M \cdot \backprime V \;=\; m \cdot v' + M \cdot V' \\ &\wedge \quad \frac{m \cdot \backprime v^2}{2} + \frac{M \cdot \backprime V^2}{2} = \frac{m \cdot v'^2}{2} + \frac{M \cdot V'^2}{2} \quad . \end{aligned} \tag{27}$$

Letting $a := M/m$ and discarding the trivial case $v' = \backprime v$ and $V' = \backprime V$, calculation yields

$$R\,(\backprime s, s') \quad \equiv \quad v' - \backprime v = a \cdot (\backprime V - V') \;\wedge\; v' + \backprime v = \backprime V + V' \quad .$$

Considering $R\,(\backprime s, s')$ as an equation with unknown s' for given $\backprime s$, we obtain

$$R\,(\backprime s, s') \quad \equiv \quad v' = -\frac{a-1}{a+1} \cdot \backprime v + \frac{2 \cdot a}{a+1} \cdot \backprime V \;\wedge\; V' = \frac{2}{a+1} \cdot \backprime v + \frac{a-1}{a+1} \cdot \backprime V \quad . \tag{28}$$

We consider two particular cases, assuming $\backprime s = w, 0$ (state just before the collision).

- Case $a = 1$. Then $v' = 0$ and $V' = w$, so $s' = 0, w$ (the balls "switch roles").
- Case $a = 3$. Then $v' = -w/2$ and $V' = w/2$, hence $s' = -w/2, w/2$. Assuming the collision took place at the lowest position, the balls move to the same height h (with $h = \frac{w^2}{8 \cdot g}$), and return to collide at the same spot with $\backprime s = w/2, -w/2$, for which (28) yields $s' = -w, 0$. Starting with the next collision, the cycle repeats.

The crux is that mathematics is not used as just a "compact language" (a layman's view of mathematics), but that the calculations yield insights hard to obtain by intuition.

5.2.2 Program equations

The *state* s is the tuple made of the program variables, and $\mathbf{S}$ its type. We let $\backprime s$ denote the state before and s' after executing a command. This allows referring to different states in one equation. We write $s \bullet e$ for $s : \mathbf{S} \,.\, e$.

If C is the set of commands, $\mathsf{R} : C \to \mathbf{S}^2 \to \mathbb{B}$ and $\mathsf{T} : C \to \mathbf{S} \to \mathbb{B}$ are defined such that the effect of a command c can be described by two equations: $\mathsf{R}\,c\,(\backprime s, s')$ for state change and $\mathsf{T}\,c\,\backprime s$ for termination. In fixed contexts, when it reduces renaming we either use s for $\backprime s$, writing $\mathsf{R}\,c\,(s, s')$ and $\mathsf{T}\,c\,s$, or use s for s', writing $\mathsf{R}\,c\,(\backprime s, s)$ and $\mathsf{T}\,c\,\backprime s$.

Our running example will be Dijkstra's *guarded command* language [14].

Command c	State change $\mathsf{R}\,c\,(s, s')$	Termination $\mathsf{T}\,c\,s$
$v := e$	$s' = s[^v_e]$	1
`skip`	$s' = s$	1
`abort`	0	0
$c' \,;\, c''$	$\exists\, t \bullet \mathsf{R}\,c'\,(s, t) \wedge \mathsf{R}\,c''\,(t, s')$	$\mathsf{T}\,c'\,s \wedge \forall\, t \bullet \mathsf{R}\,c'\,(s, t) \Rightarrow \mathsf{T}\,c''\,t$
`if` $[]\; i : I \,.\, b_i$ `->` c'_i `fi`	$\exists\, i : I \,.\, b_i \wedge \mathsf{R}\,c'_i\,(s, s')$	$\exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{T}\,c'_i\,s$

The loop `do` b `->` c' `od` by definition stands for `if` $\neg\, b$ `->` `skip` `[]` b `->` $(c' \,;\, c)$ `fi`, where c is the loop command itself.

5.2.3 Deriving the formulas of axiomatic semantics

The formulas that serve as axioms in the well-known axiomatic semantics of Hoare [19] and Dijkstra [13] can be derived as theorems from the program equations.

Hoare semantics Let the state before and after executing c satisfy a (*antecondition*) and p (*postcondition*) respectively. We write B for the set of such propositions.

Since all that is known about $\backslash s$ and s' is $a[^s_{\backslash s}$ and $\mathsf{R}\,c\,(\backslash s, s')$, this must imply $p[^s_{s'}$. This is the intuition behind the definitions of the following correctness criteria:

$$\begin{array}{lrcll}
\text{Partial correctness:} & \{a\}\ c\ \{p\} & \equiv & \forall \backslash s \bullet \forall\, s' \bullet a[^s_{\backslash s} \wedge \mathsf{R}\,c\,(\backslash s, s') \Rightarrow p[^s_{s'} & (29)\\
\text{Termination:} & \mathit{Term}\,c\,a & \equiv & \forall\, s \bullet a \Rightarrow \mathsf{T}\,c\,s & (30)\\
\text{Total correctness:} & [a]\ c\ [p] & \equiv & \{a\}\ c\ \{p\} \wedge \mathit{Term}\,c\,a & (31)
\end{array}$$

Deriving the axioms and inference rules as (meta)theorems is rather straightforward. We leave this as exercises, and provide a calculation example only for the next topic.

Calculating Dijkstra semantics We say that proposition q' is weaker than q, written $q' \sqsubseteq q$, iff $\forall\, s \bullet q \Rightarrow q'$. Hence for given p the weakest antecondition wa for total correctness is characterized by $\forall\, a : B \,.\, [a]\ c\ [p] \equiv wa \sqsubseteq a$ (uniqueness is easy to prove).

So we define the weakest liberal antecondition operator wla and the weakest antecondition operator wa by $\{a\}\ c\ \{p\} \equiv \forall\, s \bullet a \Rightarrow \mathsf{wla}\,c\,p$ and $[a]\ c\ [p] \equiv \forall\, s \bullet a \Rightarrow \mathsf{wa}\,c\,p$. To obtain explicit formulas, we transform $[a]\ c\ [p]$ into this shape by calculation.

$$\begin{array}{rcll}
[a]\ c\ [p] & \equiv & \langle\text{Definit. (31)}\rangle & \{a\}\ c\ \{p\} \wedge \mathit{Term}\,c\,a\\
& \equiv & \langle\text{Def. (29,30)}\rangle & \forall\,(\backslash s \bullet \forall\, s' \bullet a[^s_{\backslash s} \wedge \mathsf{R}\,c\,(\backslash s, s') \Rightarrow p[^s_{s'}) \wedge \forall\,(s \bullet a \Rightarrow \mathsf{T}\,c\,s)\\
& \equiv & \langle\text{Repl. } \backslash s \text{ by } s\rangle & \forall\,(s \bullet \forall\, s' \bullet a \wedge \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}) \wedge \forall\,(s \bullet a \Rightarrow \mathsf{T}\,c\,s)\\
& \equiv & \langle\text{Distr. } \forall/\wedge\rangle & \forall\, s \bullet \forall\,(s' \bullet a \wedge \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}) \wedge (a \Rightarrow \mathsf{T}\,c\,s)\\
& \equiv & \langle\text{Shunt } \wedge/\Rightarrow\rangle & \forall\, s \bullet \forall\,(s' \bullet a \Rightarrow \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}) \wedge (a \Rightarrow \mathsf{T}\,c\,s)\\
& \equiv & \langle\text{Ldist. } \Rightarrow/\forall\rangle & \forall\, s \bullet (a \Rightarrow \forall\, s' \bullet \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}) \wedge (a \Rightarrow \mathsf{T}\,c\,s)\\
& \equiv & \langle\text{Ldist. } \Rightarrow/\wedge\rangle & \forall\, s \bullet a \Rightarrow \forall\,(s' \bullet \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}) \wedge \mathsf{T}\,c\,s
\end{array}$$

Note the similarity with the ad-calculations. We proved: $\mathsf{wla}\,c\,p \equiv \forall\, s' \bullet \mathsf{R}\,c\,(s, s') \Rightarrow p[^s_{s'}$ and $\mathsf{wa}\,c\,p \equiv \mathsf{wla}\,c\,p \wedge \mathsf{T}\,c\,s$. Substituting the program equations for the various constructs, calculation in our predicate calculus yields the following results from [14].

$$\begin{array}{rcl}
\mathsf{wa}\,[\![v := e]\!]\,p & \equiv & p[^v_e\\
\mathsf{wa}\,[\![c' \,;\, c'']\!]\,p & \equiv & \mathsf{wa}\,c'\,(\mathsf{wa}\,c''\,p)\\
\mathsf{wa}\,[\![\texttt{if}\ \ [\!]\, i : I \,.\, b_i \texttt{ -> } c'_i\ \ \texttt{fi}]\!]\,p & \equiv & \exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{wa}\,c'_i\,p\\
\mathsf{wa}\,[\![\texttt{do } b \texttt{ -> } c' \texttt{ od}]\!]\,p & \equiv & \exists\, n : \mathbb{N} \,.\, w^n\,(\neg\, b \wedge p)\\
\text{defining } w \text{ by } \ w\,q & \equiv & (\neg\, b \wedge p) \vee (b \wedge \mathsf{wa}\,c'\,q) \quad .
\end{array}$$

Details are given in [10], where also the duals for strongest postconditions are derived.

Practical rules for loops Clearly $\mathsf{wa}\,[\![\texttt{do } b \texttt{ -> } c' \texttt{ od}]\!]\,p \equiv \exists\, n : \mathbb{N} \,.\, w^n\,(\neg\, b \wedge p)$ is impractical for calculation. Loop invariants and bound expressions are more convenient.

Let D be a set with order $<$ and $W : \mathcal{P}\,D$ a well-founded subset under $<$. Then an expression e of type D is a *bound expression* for c iff (i) $\forall\, s \bullet b \Rightarrow e \in W$; (ii)

$\forall\, w : W \,.\, [b \wedge w = e]\ c'\ [e < w]$. Combining:

$$\text{DEFINITION: } i : \mathbb{B} \text{ and } e : D \text{ are an } \textit{invariant/bound pair} \text{ for } c \text{ iff}$$
$$\text{(i) } \forall\, s \bullet i \wedge b \Rightarrow e \in W \quad \text{and} \quad \text{(ii) } \forall\, w : W \,.\, [i \wedge b \wedge w = e]\ c'\ [i \wedge e < w] \tag{32}$$

As in [10], we can prove the following theorem.

$$\text{THEOREM: If } i, e \text{ is an invariant/bound pair for } c \text{ then } [i]\ c\ [i \wedge \neg\, b] \tag{33}$$

Using (33) in practice is best done via a *checklist*, as suggested by Gries [17]: to show $[a]$ `do` b `->` c' `od` $[p]$, find suitable i, e and prove
(i) i satisfies $[i \wedge b]\ c'\ [i]$ or, equivalently, $i \wedge b \Rightarrow \mathsf{wa}\, c'\, i$.
(ii) i satisfies $a \Rightarrow i$.
(iii) i satisfies $i \wedge \neg\, b \Rightarrow p$.
(iv) e satisfies $i \wedge b \Rightarrow e \in W$.
(v) e satisfies $[i \wedge b \wedge w = e]\ c'\ [e < w]$ or $i \wedge b \wedge w = e \Rightarrow \mathsf{wa}\, c'\, (e < w)$ for any $w : W$.
Heuristics for finding i are (a) writing p as a conjunct and taking one as i, the negation of the other as b; (b) making a constant parameter of the problem into a variable.

6 Examples III: common aspects

Automata theory is a classical common ground between computing and systems theory. Yet, even here formalization yields unification and new insights. The example is sequentiality (capturing non-anticipatory behavior) and the derivation of properties by predicate calculus.

Preliminaries For set A, define A^n by $A^n = \square\, n \to A$ where $\square\, n = \{m : \mathbb{N} \mid m < n\}$ for $n : \mathbb{N}$ or $n := \infty$, e.g., $(0, 1, 1, 0) \in \mathbb{B}^4$. Also, $A^* = \bigcup n : \mathbb{N} \,.\, A^n$ (lists). The concatenation operator is $+\!\!+$, e.g., $(0, 7, e) +\!\!+ (3, d) = 0, 7, e, 3, d$. Also, $x \prec a = x +\!\!+ \tau\, a$. Next we consider systems $s : A^* \to B^*$.

Causal systems We define *prefix ordering* $\leq$ on A^* (and similarly for B^*) by

$$x \leq y \ \equiv\ \exists\, z : A^* \,.\, y = x +\!\!+ z\ ,$$

Definition: a system s is *sequential* iff $x \leq y \Rightarrow s\, x \leq s\, y$. This captures the intuitive notion of causal (better: "non-anticipatory") behavior. Function $r : (A^*)^2 \to B^*$ is a *residual behavior* (rb) function for s iff $s\, (x +\!\!+ y) = s\, x +\!\!+ r\, (x, y)$. We show:

THEOREM: s is sequential iff it has an rb function.

Proof: we start from the sequentiality side.

$$\begin{array}{ll}
\forall\, (x, y) : (A^*)^2 \,.\, x \leq y \Rightarrow s\, x \leq s\, y & \\
\quad \equiv \langle \text{Definit. } \leq \rangle & \forall\, (x, y) : (A^*)^2 \,.\, \exists\, (z : A^* \,.\, y = x +\!\!+ z) \Rightarrow \exists\, (u : B^* \,.\, s\, y = s\, x +\!\!+ u) \\
\quad \equiv \langle \text{Rdst } \Rightarrow/\exists \rangle & \forall\, (x, y) : (A^*)^2 \,.\, \forall\, (z : A^* \,.\, y = x +\!\!+ z \Rightarrow \exists\, u : B^* \,.\, s\, y = s\, x +\!\!+ u) \\
\quad \equiv \langle \text{Nest, swp} \rangle & \forall\, x : A^* \,.\, \forall\, z : A^* \,.\, \forall\, (y : A^* \,.\, y = x +\!\!+ z \Rightarrow \exists\, u : B^* \,.\, s\, y = s\, x +\!\!+ u) \\
\quad \equiv \langle \text{1-pt, nest} \rangle & \forall\, (x, z) : (A^*)^2 \,.\, \exists\, u : B^* \,.\, s\, (x +\!\!+ z) = s\, x +\!\!+ u \\
\quad \equiv \langle \text{Compreh.} \rangle & \exists\, r : (A^*)^2 \to B^* \,.\, \forall\, (x, z) : (A^*)^2 \,.\, s\, (x +\!\!+ z) = s\, x +\!\!+ r\, (x, z)
\end{array}$$

This completes the proof. Remarkably, the definition of $+\!\!+$ is used nowhere, illustrating the power of abstraction.

The last step uses the *function comprehension* axiom: for any relation $R : X \times Y \to \mathbb{B}$, we have $\forall\, (x : X \,.\, \exists\, y : Y \,.\, R\, (x, y)) \ \equiv\ \exists\, f : X \to Y \,.\, \forall\, x : X \,.\, R\, (x, f\, x)$.

Derivatives and primitives This framework leads to the following. An rb function is unique (exercise). We define the *derivative* operator D on sequential systems by $s\,(x \prec a) = s\,x +\!\!+ \mathrm{D}\,s\,(x \prec a)$, so $\mathrm{D}\,s\,(x \prec a) = r\,(x, \tau\,a)$ where r is the rb function of s, and by $D\,s\,\varepsilon = \varepsilon$.

Primitivation I is defined for any $g : A^* \to B^*$ by $\mathrm{I}\,g\,\varepsilon = \varepsilon$ and $\mathrm{I}\,g\,(x \prec a) = \mathrm{I}\,g\,x +\!\!+ g\,(x +\!\!+ a)$. Properties are shown next, with a striking analogy from analysis.

$s\,(x \prec a) = s\,x +\!\!+ \mathrm{D}\,s\,(x \prec a)$	$s\,x = s\,\varepsilon +\!\!+ \mathrm{I}\,(\mathrm{D}\,s)\,x$
$f\,(x+h) \approx f\,x + \mathrm{D}\,f\,x \cdot h$	$f\,x = f\,0 + \mathrm{I}\,(\mathrm{D}\,f)\,x$

Of course, in the second row, D is the derivation operator from analysis, and $\mathrm{I}\,g\,x = \int_0^x g\,y \cdot \mathrm{d}\,y$ for integrable g. Moreover, $f\,(x+h) = f\,x + \mathrm{D}\,f\,x \cdot h$ is only approximate.

This and other differences confirm the observation in [25] that automata are easier than real functions.

Finally, $\{(y : A^*\,.\,r\,(x, y)) \mid x : A^*\}$ is the *state space*.

7 Conclusion

We have shown how a formalism, consisting of a very simple language of only 4 constructs, together with a powerful set of formal calculation rules, not only yields a notational and methodological unification of computing science and classical engineering, but also of a large part of mathematics.

Apart from the obvious scientific ramifications, the formalism provides a unified basis for education in ECE (Electrical and Computer Engineering), as advocated by Lee and Messerschmitt [24].

In such a curiculum, a course in formal calculation (in particular with generic functionals, predicates and quantifiers) would precede most other courses in engineering mathematics (including mathematical analysis). This provides an opportunity for the latter courses to consolidate and exercise the students' abilities in calculational logic, thereby providing an equally solid basis for computing science courses.

References

[1] Henk P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North-Holland (1984)

[2] Richard E. Blahut, *Theory and Practice of Error Control Codes*. Addison-Wesley (1984)

[3] Eerke Boiten and Bernhard Möller, *Sixth International Conference on Mathematics of Program Construction* (Conference announcement), Dagstuhl (2002). `www.cs.kent.ac.uk/events/conf/2002/mpc2002`

[4] Raymond T. Boute, "A heretical view on type embedding", *ACM Sigplan Notices 25*, pp. 22–28 (Jan. 1990)

[5] Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen (July 1993)

[6] Raymond T. Boute, "Fundamentals of hardware description languages and declarative languages", in: Jean P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, pp. 3–38, Kluwer Academic Publishers (1993)

[7] Raymond T. Boute, "Supertotal Function Definition in Mathematics and Software Engineering", *IEEE Transactions on Software Engineering, Vol. 26*, No. 7, pp. 662–672 (July 2000)

[8] Raymond Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)

[9] Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons and Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)

[10] Raymond T. Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus", Technical note B2004/02, INTEC, Universiteit Gent (2004) (submitted for publication to *ACM TOPLAS*)

[11] Ronald N. Bracewell, *The Fourier Transform and Its Applications*, 2nd ed, McGraw-Hill (1978)

[12] Ralph S. Carson, *Radio Communications Concepts: Analog*. Wiley (1990)

[13] Edsger W. Dijkstra, *A Discipline of Programming*. Prentice-Hall (1976)

[14] Edsger W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin (1990)

[15] Edsger W. Dijkstra, "Under the spell of Leibniz's dream", *EWD1298* (April 2000).

[16] Ganesh Gopalakrishnan, *Computation Engineering: Formal Specification and Verification Methods* (Aug. 2003).
`http://www.cs.utah.edu/classes/cs6110/lectures/CH1/ch1.pdf`

[17] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, Berlin (1993)

[18] David Gries, "The need for education in useful formal logic", *IEEE Computer 29*, 4, pp. 29–30 (April 1996)

[19] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Comm. ACM, 12*, 10, pp. 576–580, 583 (Oct. 1969)

[20] C. A. R. Hoare, He Jifeng, *Unifying Theories of Programming*. Prentice-Hall (1998)

[21] Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*. Springer-Verlag, Berlin (1978)

[22] Serge Lang, *Undergraduate Analysis*. Springer-Verlag, Berlin (1983)

[23] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education Inc. (2002)

[24] Edward A. Lee and David G. Messerschmitt, "Engineering — an Education for the Future", *IEEE Computer, Vol. 31,* No. 1, pp. 77–85 (Jan. 1998), via `ptolemy.eecs.berkeley.edu/publications/papers/98/`

[25] Edward A. Lee and Pravin Varaiya, "Introducing Signals and Systems — The Berkeley Approach", *First Signal Processing Education Workshop*, Hunt, Texas (Oct. 2000), via `ptolemy.eecs.berkeley.edu/publications/papers/00/`

[26] Jacques Loeckx and Kurt Sieber, *The Foundations of Program Verification*. Wiley-Teubner (1984)

[27] Bertrand Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall, New York (1991)

[28] Rex Page, *BESEME: Better Software Engineering through Mathematics Education*, project presentation `http://www.cs.ou.edu/~beseme/besemePres.pdf`

[29] David L. Parnas, "Education for Computing Professionals", *IEEE Computer 23*, 1, pp. 17–20 (January 1990)

[30] David L. Parnas, "Predicate Logic for Software Engineering", *IEEE Trans. SWE 19*, 9, pp. 856–862 (Sept. 1993)

[31] William Pugh, "Counting Solutions to Presburger Formulas: How and Why", *ACM SIGPLAN Notices 29*, 6, pp. 121–122 (June 1994)

[32] David A. Schmidt, *Denotational Semantics*. Wm. C. Brown Publishers, Dubuque (1988)

[33] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Massachusetts (1977)

[34] Paul Taylor, *Practical Foundations of Mathematics* (second printing), No. 59 in *Cambridge Studies in Advanced Mathematics*, Cambridge University Press (2000); quoted from the introduction to Chapter 1 in `www.dcs.qmul.ac.uk/~pt/Practical_Foundations/html`

[35] Robert D. Tennent, *Semantics of Programming Languages*. Prentice-Hall (1991).

[36] Jeannette M. Wing, "Weaving Formal Methods into the Undergraduate Curriculum", *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)* pp. 2–7 (May 2000); file `amast00.html` in `www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/`

[37] Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge (1993)